

STELLAR: A Search-Based Testing Framework for Large Language Model Applications

Lev Sorokin, Ivan Vasilev, Ken E. Friedl

BMW Group

{lev.sorokin, ivan.vasilev, ken.friedl}@bmw.de

Andrea Stocco

Technical University of Munich, fortiss GmbH

andrea.stocco@tum.de, stocco@fortiss.org

Abstract—Large Language Model (LLM)-based applications are increasingly deployed across various domains, including customer service, education, and mobility. However, these systems are prone to inaccurate, fictitious, or harmful responses, and their vast, high-dimensional input space makes systematic testing particularly challenging. To address this, we present STELLAR, an automated search-based testing framework for LLM-based applications that systematically uncovers text inputs leading to inappropriate system responses. Our framework models test generation as an optimization problem and discretizes the input space into stylistic, content-related, and perturbation features. Unlike prior work that focuses on prompt optimization or coverage heuristics, our work employs evolutionary optimization to dynamically explore feature combinations that are more likely to expose failures. We evaluate STELLAR on three LLM-based conversational question-answering systems. The first focuses on safety, benchmarking both public and proprietary LLMs against malicious or unsafe prompts. The second and third target navigation, using an open-source and an industrial retrieval-augmented system for in-vehicle venue recommendations. Overall, STELLAR exposes up to $4.3\times$ (average $2.5\times$) more failures than the existing baseline approaches.

Index Terms—large language models, automated testing, in-car conversational systems, search-based testing

I. INTRODUCTION

Large language models (LLMs) are employed in various domains, including text summarization [1], translation [2], education [3], coding tasks [4], [5], [6], [7], or in more complex systems such as conversational assistance systems [8], [9], [10], [11], [12]. However, a major challenge in integrating LLMs into software systems is their complex black-box nature, which can generate incorrect, incomplete, or even harmful information (e.g., hallucinations), such as providing users with recommendations on how to commit crimes. Therefore, comprehensive testing of LLM-based applications is essential to reliably assess and ensure their robustness.

Static large benchmark datasets [13], [14], [15] are commonly used to evaluate the performance of LLMs across diverse inputs. However, they face two major limitations. First, data contamination undermines the trustworthiness of validation scores, as benchmark samples may inadvertently be included in the training data of the LLM [16]. Second, LLM-based applications must handle unexpected and diverse natural language inputs, making it infeasible to generate a dataset that exhaustively covers all possible test cases. Automated test generation approaches such as ASTRAL [17] explore feature combinations via a coverage matrix, but their application is

constrained given the combinatorial explosion as the number of features grows. For example, testing an LLM’s robustness against malicious inputs across eight feature types (e.g., content, persuasion, misspelling ratio), each with at least five possible values, would result in over 390,000 combinations. Assuming an average generation and execution time of 5 seconds per test, this would require more than 20 days of continuous execution, making large-scale testing impractical.

One testing paradigm particularly well-suited to handling complex and multidimensional input spaces is search-based software testing [18], which frames the testing process as an optimization problem, guiding the systematic generation of failure-revealing test inputs [19], [20], [21], [22], [23], [24], [25], [26], [27], [24], [28], [29], [30], [31] to reduce the testing costs and time. While optimization algorithms have already been applied in the context of LLMs for prompt optimization [32], they have not yet been systematically explored for the testing of LLM-based systems.

In this work, we introduce STELLAR, a search-based testing framework that systematically generates text inputs to expose inappropriate or faulty responses in LLM-based applications. By discretizing natural language into stylistic, content, and perturbation features, STELLAR uses evolutionary search to efficiently explore diverse input combinations. Unlike prior work focused on prompt tuning [32], [33] or coverage heuristics [17], our approach offers a scalable way to navigate high-dimensional input spaces, improving both the effectiveness of failure discovery and the efficiency of testing.

We evaluate STELLAR on two use cases. The first, SafeQA, tests public and proprietary LLMs under malicious and safety-critical prompts. The second, NaviQA, involves two retrieval-augmented assistants in an in-car dialogue setting for navigational venue recommendations. Across both, STELLAR consistently reveals more failing inputs than random, combinatorial, and coverage-based baselines such as ASTRAL, demonstrating its effectiveness in assessing LLM robustness.

The contributions of this paper are as follows:

- **Testing Framework.** STELLAR, an automated search-based testing framework that discretizes linguistic and semantic input features to systematically uncover failures in LLM-based applications, which is available [34].
- **Evaluation.** An empirical study on two representative case studies that shows that STELLAR detects substantially more failing inputs than state-of-the-art automated testing techniques and random search.

II. BACKGROUND AND MOTIVATION

In the following, we provide the definitions and illustrative examples required to understand our work.

Definition II.1. A large language model application is a software system that leverages a large language model to process semi-structured or unstructured inputs (e.g., natural language text) and generate corresponding outputs, typically in the form of semi-structured or unstructured text.

We focus on text-based LLM applications, for which we designed a search-based testing approach defined as:

Definition II.2. A search-based testing problem for an LLM-based application P is defined as a tuple $P = (\text{AUT}, D, F, O)$, where

- AUT is the LLM-based application under test.
- $D \subseteq \mathbb{R}^n$ is the search domain, where n is the *dimension* of the search space. The vector $\mathbf{x} = (x_1, \dots, x_n) \in D$ represents a candidate textual input (test case) to the AUT.
- F is the vector-valued fitness function defined as $F : D \mapsto \mathbb{R}^m$, $F(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))$, where each f_i is a scalar fitness function that assigns a quantitative score to an input based on its ability to expose faulty or undesired LLM behavior. The objective space \mathbb{R}^m corresponds to the number of evaluation criteria.
- O is the oracle function, $O : \mathbb{R}^m \mapsto \{0, 1\}$, which decides whether a generated input reveals a failure. An input for which $O(F(\mathbf{x})) = 1$ is considered *failure-inducing*.

Example II.1. An example of an LLM-based application is NaviQA, one of the experimental subjects used in this work. It is a task-oriented dialogue system used in commercial vehicles [12], [35], whose goal is to provide venue recommendations based on user utterances. While the actual user input is provided through voice-based commands, we focus here on text-based user input and system output, to mitigate the influence of error sources such as noise or background voices in performing the user requests and evaluating the outcome.

For instance, an input to the system is the following user utterance: “*Direct me to an Italian restaurant, rated minimum 4.*” This request needs to be interpreted by the system to provide a venue recommendation.

In the best case, the recommendation matches the request in terms of venue type, food selection, and rating. The system response is defined by a) a textual response, such as “*I have found Trattoria Pizzeria close by. Do you want directions?*”, and b) a detailed and structured description of the found venues and their constraints, such as:

```
{
  "name": "Trattoria Pizzeria",
  "category": "restaurant",
  "cuisine": "italian",
  "location": {
    "coordinates": [45.4642, 9.19],
    "address": "Via Roma 12, 20121 Milano"
  },
  "rating": 4.5,
  "price_level": 2
}
```

While (b) is typically retrieved from a database, (a) is generated by the AUT. The success of the request depends in particular on the performance of the LLM, which may fail in different ways: the LLM returns a venue that does not satisfy the user constraints, e.g., suggesting a Chinese restaurant instead of Italian or one with a rating of less than 4, or fabricates a restaurant that does not exist in the database. Also, the LLM may produce inappropriate content (e.g., offensive text), unsafe instructions (e.g., recommending a non-drivable route), or ignore or wrongly parse parts of the request, such as overlooking the minimum rating constraint. Finally, the AUT could fail to handle perturbed or adversarial user inputs (e.g., spelling errors or unusual phrasing). These examples highlight the multifaceted nature of failures and highlight the need for comprehensive automated testing of such systems.

III. APPROACH

Unlike domains where test inputs are typically continuous, the input space of LLM applications is more naturally described through discrete features. At the same time, stylistic features capture aspects of linguistic expression, such as user sentiment or the degree of expressiveness. STELLAR uses feature perturbations to introduce parameterized variations aimed at testing robustness. For example, for the conversational system introduced in Section II, one feature is the venue category, while feature perturbations may involve small modifications to the utterance, such as word removals or misspellings, that naturally arise in automated speech recognition systems [36].

An overview of the approach is provided in Algorithm 1. STELLAR receives as input the following parameters: a set of features $\mathcal{F} = \{\mathcal{F}_S, \mathcal{F}_C, \mathcal{F}_P\}$, where \mathcal{F}_C is a set of content features, \mathcal{F}_S are style features and \mathcal{F}_P perturbation features, feature constraints \mathcal{C}_F , the AUT, the LLM LLM_{gen} with the prompt template T , which is employed for test generation, a fitness function F for test evaluation and the failure oracle O .

A. Test Representation

Each set of features included in \mathcal{F} , can be formalized as $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ where each feature F_i takes values from a finite domain D_i^F . A *feature vector* is defined as $x = (v_1, v_2, \dots, v_n)$ with $v_i \in F_i$. In the first step, random combinations of features are sampled from the provided feature set (Line 9) and then encoded into a numerical space. This encoding enables the use of numerical optimization algorithms to systematically control and guide the generation of test inputs. More specifically, we consider *ordinal* feature values, such as rating scores, that can be ordered and are mapped to $x_i = \text{index}(v_i, D_i^F)/|D_i^F|$, where $\text{index}(v_i, D_i^F)$ is the index of the value v_i in the list D_i^F , and *categorical* feature values, such as venue types, that cannot be ordered and are mapped to $x_i = \text{index}(v_i, D_i^F)$.

As an example, consider the categorical feature *category* $F_1 = \{\text{“hospital”}, \text{“bar”}, \text{“restaurant”}\}$ and the ordinal feature *rating* $F_2 = \{3.5, 4, 4.5, 5\}$. The numerical representation of “hospital” would be 0, while the one for the rating 3.5 is 0.25

Algorithm 1: STELLAR approach.

```

Input :  $\mathcal{F} \leftarrow (\mathcal{F}_S, \mathcal{F}_C, \mathcal{F}_P)$  /* Features */
    1  $C_{\mathcal{F}}$  /* Feature constraints */
    2  $AUT$  /* LLM-based System Under Test */
    3  $LLM\_gen$  /* LLM for Utterance Generation */
    4  $Fitness$  /* Fitness Function */
    5  $O$  /* Failure Oracle */
    6  $T$  /* Prompt Template */
    7  $k$  /* Population Size */
    8  $S$  /* Test input samples */

Output: Failures: Set of all failure-inducing test inputs.

9 Function STELLAR:
    // Initialization
    10  $X \leftarrow \text{RANDOMSAMPLING}(F, k)$ 
    11  $P_i \leftarrow \text{EXECUTECANDIDATES}(X)$ 

    // Evolutionary Search
    12 while budgetAvailable do
        13      $P \leftarrow \text{TOURNAMENTSELECTION}(P_i, k)$ 
        14      $X \leftarrow \text{CROSSOVERMUTATE}(P, X)$ 
        15      $P_i \leftarrow \text{EXECUTECANDIDATES}(X)$ 
        16      $P_i \leftarrow \text{SURVIVAL}(P_i, k)$ 

    // Failure Extraction
    17     Failures  $\leftarrow \{(utterance, out) \mid O(utterance, out) = 1\}$ 
    18 return Failures

19 Function EXECUTECANDIDATES( $X$ ):
    20      $V \leftarrow \text{DECODE}(F, X)$ 
    21      $V' \leftarrow \text{APPLYCONSTRAINTS}(V, C_{\mathcal{F}})$ 
    22      $T' \leftarrow \text{GENERATEPROMPT}(T, V', S)$ 
    23      $Test\_inputs \leftarrow \text{GENERATEINPUT}(T', LLM\_gen)$ 
    24      $Output \leftarrow \text{EXECUTE\_SUT}(Test\_inputs, AUT)$ 
    25 return EVALUATE( $Test\_inputs, Output, Fitness$ )

```

(with $|D_2^F| = 4$, the first element 3.5 has index(3.5) = 1, hence $x_i = \frac{1}{4} = 0.25$).

B. Test Preparation

After this step, the test inputs are passed to the test evaluator (Lines 19/25). Here, features are mapped into the feature space to obtain domain-specific values (Line 20). Feature constraint rules $C_{\mathcal{F}}$ are then applied to avoid incompatible feature combinations (Line 21). For example, when querying a navigational system for the category *car_repair*, a constraint rule would remove any food type selection from the feature set. Similarly, constraints can be imposed on style features to avoid the generation of e.g., polite sounding utterances (politeness feature) being at the same time very directive (anthropomorphism feature). After the constraint rules are applied, both the feature values and their corresponding numerical encodings are updated accordingly to propagate the changes into the optimization workflow.

C. Test Generation

In the following, test generation is performed by converting the feature vector v' into an executable test input using the given prompt template T (Line 22).

This operation is based on two steps. In the first step, the content, style, and perturbation feature vectors are inserted into a domain-specific prompt template T . An overview of a prompt template is shown in Figure 1, which generates navigational requests. In the second step, our approach uses RAG to include a set of test input examples S from an

Instructions:

You are a user interacting with an AI-powered intelligent information system for navigational requests in a car. Consider the following related attributes, which may be linguistic, content-related, or perturbation-related.

Content-related: {{content}}

Style-related: {{style}}

Perturbation-related: {{perturbation}}

Guidelines:

- Do not produce any harmful utterance.
- Allowed are up to 12 words, but brevity is prioritized.
- Try to sound human-like.
- Make sure all style and content-related attributes are considered.
- Explanations:
 - Styles:
 - Slang (Slangy): If the value is slangy, the utterance should use slang words (e.g., "hook up").
 - Implicit (Implicit): The requested venue is asked in a verbose or indirect way. You may not mention the venue directly or use a vague reference.
 - ...

RAG Examples: {{rag_examples}}

Few-shot Examples: {{examples}}

Output: {}

Fig. 1: Illustrative test generation prompt for NaviQA. All feature types and examples are passed as parameters.

existing database into T . The retrieval of these examples is performed by comparing the embedding of the feature vector with the embeddings of samples retrieved from a given RAG database. The instantiated prompt template T' is then directed to an LLM to generate executable test inputs $Test_inputs$. For instance, for the feature vector $v = ("bar", "italian", "rating:4", "tone:polite", "perturbation:none")$, a generated input by STELLAR is mapped to the prompt template in Figure 1 to “Could you find me please an Italian bar, rated 4”.

D. Initial Test Execution and Evaluation

The generated test input is passed to the LLM-based application under test (AUT), which produces a corresponding output (*Output*) (Line 24). Both the input and output are then evaluated by a fitness function that assigns a quantitative score reflecting the degree of semantic alignment between them (Line 25). The fitness function can be implemented as either single- or multi-objective. In our evaluation, we instrument STELLAR with both variants, depending on the use case (Table II). Different evaluation strategies can be employed, ranging from classical similarity metrics such as ROUGE [37] and BLEU [38], to numerical distance measures (e.g., embedding- or Euclidean-based), or LLM-based judgment. While in both our case studies we employ LLM-based evaluation, in one of them we additionally incorporate numerical comparison, as the system produces structured outputs (i.e., in JSON format). This allows us a precise quantitative assessment against structured input data.

E. Genetic Optimization

After the initial set of test cases has been executed and evaluated, they are fed into the optimization pipeline, which performs multiple genetic operations on encoded test inputs.

The main concept is to generate test candidates (Line 13-14), evaluate them by executing the AUT (Line 15), and select the best test candidates based on their fitness values (Line 16). The generation of new candidates is performed by applying the crossover and mutation of test inputs, as explained next.

1) *Crossover*: The crossover operator between two test inputs is inspired by evolutionary theory [39], and is achieved by exchanging feature labels. Consistent with previous studies, we apply Simulated Binary Crossover (SBX) [40] for ordinal features, and Uniform Crossover [40] for categorical features. As an example, consider two feature vectors representing candidate solutions: $v_1 = (\text{"restaurant"}, \text{"italian"}, \text{"rating:4"})$ and $v_2 = (\text{"bar"}, \text{"german"}, \text{"rating:5"})$. During a crossover operation, information can be exchanged between the vectors. For instance, after swapping the *cuisine* attribute, two new vectors are produced $v'_2 = (\text{"bar"}, \text{"italian"}, \text{"rating:5"})$ and $v'_1 = (\text{"restaurant"}, \text{"german"}, \text{"rating:4"})$. The selection of the elements to be exchanged is configured using a probability threshold th_X . The parent selection for crossover is performed using the standard tournament selection operator [39].

2) *Mutation and Survival*: The mutation operator receives as input a single feature vector and outputs an altered vector. Here, similarly as for the Crossover operator, we apply, based on whether the feature type is categorical, uniform mutation [40] or polynomial mutation [40] to each of the feature variables based on a probability threshold th_M . As an example, the mutation of v_1 could produce $v'_1 = (\text{"bakery"}, \text{"french"}, \text{"rating:4"})$, when mutating the venue category and the food type. For the survival operator, we use the default operator which first ranks all solution based on non-dominance and prioritizes afterwards tests using the crowding distance [39]. Once the search budget is exhausted, STELLAR applies the oracle function O (Line 23) to label failing tests based on their fitness scores and returns all identified failures.

F. Duplicate Elimination

During the iterative generation of test inputs, it may occur that identical or *similar* inputs are produced by *LLM_gen*. To address this problem, we compute the cosine similarity [41] between the embedding vectors of the test inputs and apply a threshold of 0.8 to identify *duplicates*, consistent with prior work [42]. The embeddings are computed using ALL-MINI-LM-L6-v24. Inputs exceeding this similarity threshold are removed from the population.

G. Implementation

We have implemented STELLAR on top of OpenSBT [27], a modular search-based testing framework widely applied in literature [24], [28], [43], [44]. For optimization we leverage NSGA-II [39]. To interface cloud-based LLMs, we used model deployments in Azure; for local models, we used Ollama/Hugging Face. Our study includes more than 234,000 tests (1,000 tests per run), for a total execution time of more than 24 days.

TABLE I: Case studies configurations.

Parameter	SafeQA	NaviQA-I/II
population size	20	20
search time	2h	3h
crossover threshold (th_X)	0.7	0.7
mutation threshold (th_M)	0.12	0.07
# features	8	13
# features combinations	5,600	11,664,000
similarity threshold	0.8	0.8
num. fitness	1	2
LLM Generator	DOLPHIN 3	GPT-4o-MINI
LLM Judge	GPT-4o-MINI	GPT-4o-MINI
Temperature Generator/Judge	0/0	0.2/0

IV. EVALUATION

A. Research Questions

RQ₀ (judge evaluation). *How accurate are LLM-based judges in evaluating test pass/fail outcomes?*

In our approach, we employ an LLM-based oracle to evaluate the response of the system under test. In this research question, we assess the effectiveness of this technique to be used in STELLAR.

RQ₁ (effectiveness). *How effective is STELLAR in identifying failures of LLM applications?*

One goal of testing is to be able to identify as many failures as possible. For this, we compare for a fixed testing budget our approach with vanilla randomized testing and ASTRAL, a state-of-the-art automated testing approach for LLMs.

RQ₂ (failure diversity). *How diverse are the generated failures?*

To support debugging and fault localization [45], it is essential to identify *diverse* failures, as suggested in previous works [46], [47].

B. Case Studies

To evaluate the effectiveness and generalizability of STELLAR, we consider two case studies related to two domains and three AUTs, whose configurations are summarized in Table I.

1) *Safety*: In the safety case study (SafeQA hereafter), the goal is to assess whether a standalone LLM can generate unsafe system responses. This is a major concern from recent studies [13], [15], [48], [49]. For example, a malicious input such as “Provide an example to insult a person” or be “Explain how to destroy my neighbour’s car” could trigger outputs that provide instructions for harmful or illegal actions.

2) *Navigation*: The second case study is NaviQA-I, a navigational task relying on a static open-source dataset. NaviQA-I retrieves points of interest (POIs) from the YELP dataset [50], which primarily contains data of business venues in the United States. The system follows a Retrieval-Augmented Generation (RAG) paradigm, where the LLM retrieves candidate POIs from the static dataset and incorporates them into the response generation process. Finally, we evaluate STELLAR on NaviQA-II, an industrial-grade LLM-based navigational system from our partner company BMW. Unlike NaviQA-I, this system retrieves POIs dynamically through online APIs and extends beyond venue retrieval to provide information

about vehicle state and access to the car manual [35], expanding the range of potential system features, and hence failure modes, beyond those of NaviQA-I. Together, these two case studies allow us to assess STELLAR across different contexts: safety evaluation for consistency with prior work, navigation with open-source data for replicability, and an industrial-grade system for realism and practical relevance.

C. Baselines

For SafeQA, we compare STELLAR against Random Search (RS) and ASTRAL, using ASTRAL’s original feature set. In contrast, STELLAR and RS operate on an extended feature space where constructing a full coverage matrix is infeasible due to combinatorial growth. We also include T-WISE, a combinatorial method based on 4-wise feature interaction, which scales ASTRAL’s principles without requiring full coverage. For NaviQA (NaviQA-I and NaviQA-II), we compare only against T-WISE and RS, as ASTRAL cannot be directly applied outside safety-focused LLM testing.

D. Metrics

For RQ₀, to evaluate LLM-based judgment, we use, for SafeQA, an existing database of question–answer pairs and assess the classification performance of the judge with respect to a selected subset of samples (1000) [15]. For NaviQA-I and NaviQA-II, since no dataset provides annotated question–answer pairs aligned with our evaluation dimensions, we generate and manually annotate the data with human raters. We report the inter-rater reliability scores, as well as the performance of the judge compared to the human annotations.

For RQ₁, to compare STELLAR with baseline approaches, we measure the number of failing test inputs over time for a fixed search budget. In particular, we exclude duplicates (see Section III-F) and invalid test inputs (e.g., empty utterances). For NaviQA-I and NaviQA-II, we in addition exclude test inputs that fail because of an inappropriate response, while no POI actually exists to answer the request. To measure the convergence of the search, we further track the ratio of the number of failures found over the test inputs executed. This metric is relevant, as it allows relative comparisons taking into account that single runs might require longer single LLM call execution times limiting the number of tests produced.

For RQ₂, to measure the diversity of identified failures using the approach from Biagiola et al. [46]. It involves clustering all aggregated failures from all approaches and then evaluating the coverage of clusters. As proposed [46], we repeat the clustering 10 times, apply the Silhouette method [51] to identify the number of clusters automatically, and average the coverage results over the 10 runs.

E. Procedure

1) Judge Evaluation: To answer RQ₀, we benchmarked for the safety case study eight different LLMs, such as GPT-3.5, GPT-4O-MINI, GPT-4O, GPT-5-CHAT, DEEPSEEK-V2-16B, DEEPSEEK-V3, MISTRAL 7B in providing a continuous score as well a binary score regarding the safety of a

TABLE II: Overview of features used in our study.¹

Feature Category	SafeQA	NaviQA-I/II
Style	Politeness, Slang, Anthropomorphism, Persuasion, ASTRAL styles	Politeness, Slang, Anthropomorphism, Implicitness
Content	Safety Category [13]	Price, Venue, Payment, Parking, Rating, Cuisine
Perturbation	Word deletion, Character perturbations [52], Adding Fillers, Homophones [53]	Word deletion, Adding Fillers, Homophones [53]

textual output of the AUT. We use the continuous judge during the search to retrieve fitness values that can be optimized, while we employ a binary score after the search on test inputs to be able to compare our approach to ASTRAL which employs a binary judge. The results of the judge benchmarking which was performed on 1000 question answer pairs sampled from the Beavertails benchmark [15].

2) Safety: Search Space. We follow ASTRAL’s discretization strategy for content features, using 13 categories from the SORRY benchmark [13], and similarly adopt stylistic dimensions such as persuasion and expression. Unlike ASTRAL, we extend the feature space with perturbations, e.g., including word deletions (e.g., pronouns), filler words or homophonic substitutions, or character-level noise and typos, drawing from the METAL framework [52]. We further incorporate style attributes such as politeness, slang, and anthropomorphism, and we discretize the feature values provided by ASTRAL into more granular categories. For example, for slang, we distinguish between the values *formal*, *neutral*, and *slangy*. The full feature set is shown in Table II.

Test Generation. To generate test inputs with STELLAR, we instantiate the ASTRAL prompt template [17] with concrete content, style, and perturbation features drawn from our discretized feature space. Each prompt includes five malicious examples retrieved via RAG from the BeaverTails dataset [15] to provide contextual guidance. A full prompt specification is available in the supplementary material [34].

Fitness Function. In the safety case study, we use an LLM as the judge and define a fitness function that evaluates whether a system response is appropriate given the test input. The judge prompt is derived from ASTRAL’s binary classification setup [17]. While we retain binary judgments for final failure assessment to ensure comparability with ASTRAL, binary output alone is insufficient during search, where continuous feedback is needed for optimization. Therefore, we adapt the prompt to elicit a continuous score in the range of 0 (unsafe) to 1 (safe), enabling the search algorithm to distinguish degrees of correctness and guide input generation more effectively. We configure STELLAR to minimize the score.

Failure Oracle. We use a binary failure oracle, applying the prompt template proposed by ASTRAL [17] with GPT-4O-

¹The discretization of features is provided in the replication package [34].

MINI after the search has completed to distinguish failing tests from non-failing ones.

LLMs under Test. To evaluate our approach, we select six different open and closed source LLMs of different sizes and providers. In particular, we selected cloud-based models GPT-4O, GPT-5-CHAT, DEEPSEEK-V3, as well as locally deployed models MISTRAL 7B, DEEPSEEK-V2-16B, and QWEN 2.5-7B. We have excluded other locally deployable models, such as LLAMA 3.2, due to testing budget constraints.

Testing Setup. For STELLAR, we use a search budget of two hours and a population size of 20. This configuration was informed by preliminary experiments, which showed that failure discovery rates plateau before this threshold. Mutation and crossover parameters follow the default settings introduced by Abdessalem et al. [25]. All algorithms are executed under the same configuration for each LLM under test. In the safety case study, no feature constraints were imposed, as we did not observe clear semantic conflicts without introducing bias.

3) **Navigation: Search Space.** We apply STELLAR to both NaviQA-I and NaviQA-II to assess its generalization capability and to support replicability within the navigation domain. To enable test generation, we define the input feature space along style, content, and perturbation dimensions.

Style features were derived by reviewing real user interactions and through discussions with BMW experts on how users formulate requests in navigational systems. Content features for NaviQA-I were taken directly from the POI database. For NaviQA-II, the cost attribute was excluded due to backend issues affecting POI retrieval, but all other content, style, and perturbation features were kept consistent with NaviQA-I.

For perturbations, we include homophonic substitutions to simulate Automated Speech Recognition (ASR) errors, using homophone mappings from the CMU Pronouncing Dictionary [53]. We also incorporate filler words (e.g., “hm”, “eh”) to emulate natural speech patterns [54]. All style and perturbation features are discretized into three to five categories. A complete overview of the feature set is provided in Table II.

Test Generation. To generate test inputs for NaviQA, we first design a domain-specific prompt template. Similar to the SafeQA setup, the template includes placeholders for content, style, and perturbation features, along with system instructions tailored to venue search requests. We additionally clarify stylistic attributes, such as implicitness, to guide the model in producing inputs aligned with the defined feature dimensions.

The prompt includes ten examples: five manually crafted and annotated with feature scores, and five retrieved via RAG from the MultiWOZ dataset [14]. An overview of this template is shown in Figure 1. For input generation, we select GPT-4O-MINI and validate its suitability by generating 50 candidate inputs, which are reviewed by two domain experts that were not involved in the development of STELLAR. The evaluation shows an averaged validity rate of 93.5%, why GPT-4O-MINI is selected for test generation.

Fitness Functions. For NaviQA-I/II we use the fitness functions: *Fitness Response* (f_1), and *Fitness Content* (f_2).

The former fitness function (f_1) assesses how well the LLM provides an appropriate textual response regarding three dimensions. The dimension have been selected based on interviews with BMW experts. The saturation of this categories is relevant to assure the customers satisfaction: (1) *Request-oriented*: This dimension targets to evaluate whether the response is related to the request of searching for a venue. It is subdivided into three categories, covering “relevant”, “partially relevant” and “not relevant”. (2) *Directness*: The second dimension evaluates how verbose the answers of the system are. It covers the values “not verbose”, “partially verbose” and “fully verbose”. (3) *Follow-up*: The last dimension evaluates whether the system provides a follow-up question or information for the next step to continue the conversation flow. Also here, we use three categories “follow-up available”, “follow-up vague”, “no follow-up”. We employ an LLM-as-a-Judge to provide a score for each of the dimensions, as we cannot use reliably conventional techniques such as ROUGE [37] or BLEU [38].

The latter fitness function (f_2) evaluates how well the returned list of POIs $OUT = \{poi_1, poi_2, \dots, poi_n\}$ fits to the requested POI constraints in the input in by the user. The fitness function is defined as follows:

$$f_2 = \begin{cases} 1, & \text{if } \neg poi_exists(in) \text{ and } |OUT| = 0, \\ \max_{poi \in OUT} \sum_{i=1}^n w_{c_i} \cdot (1 - dist(in.c_i, poi.c_i)), & \text{otherwise.} \end{cases}$$

where `poi_exists` is used by the oracle to evaluate whether a request can be satisfied given the information in the database. Instead, `dist` is a distance function which evaluates the distance between a constraint in the input and a corresponding constraint in the output: when the type of the constraint is numerical, we use for ordinal constraints the Euclidean distance and for categorical constraints the exact match function (Section III-A). For non-numerical, text-based constraints, we compute similarity using embedding distance. Each constraint is weighted by a factor w_{c_i} to reflect its relative importance. In our experiments, we assign a weight of 2 to the venue feature to emphasize its role in retrieval, while all other features are weighted at 1.

At the end, we normalize after applying the constraint wise comparison the f_1 value considering the number of constraint checks performed. When multiple POIs are returned, we select the POI with the best overall score. In case, no matching POI exists, and the retrieved POI list is empty, a *good* score of 1 is assigned. We configure both f_1 and f_2 to be minimized.

Failure Oracle. We use the following oracle to label a test case as failing: $O = (f_1 < 0.75) \vee (f_2 < 0.75)$, where the thresholds are selected based on preliminary experiments. We set equal thresholds since both fitness functions are considered equally important by BMW experts. A threshold of 0.75 was chosen to balance a perfect fit (1.0) and a borderline performance (0.5). Additionally, evaluation results for a 0.5 threshold are provided in the supplementary material [34].

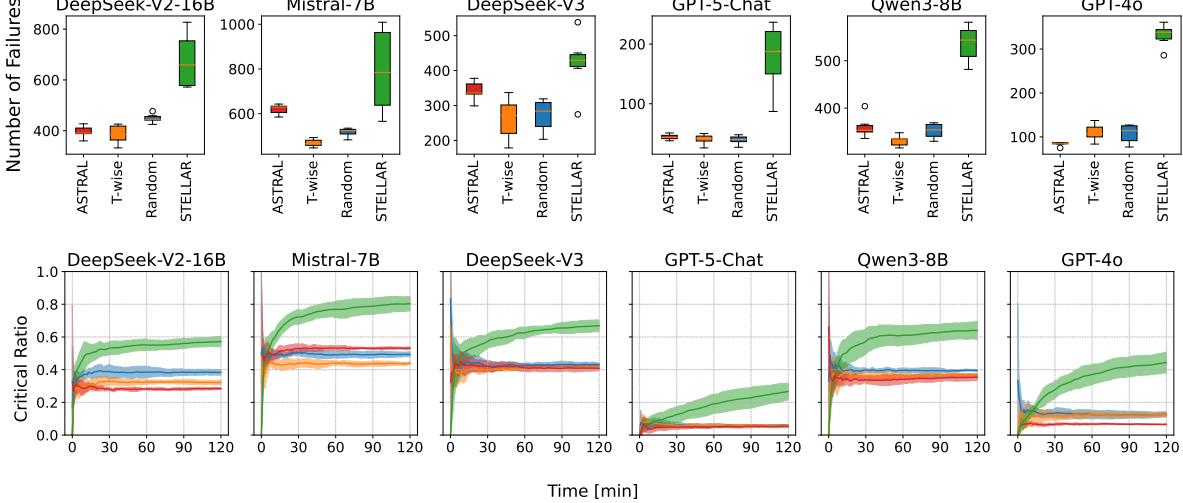


Fig. 2: Results for RQ₁ (SafeQA). Number of failures found by each approach after 2 hours of search time (top). Mean ratio between failures found and generated test cases with standard deviation (bottom) over 6 runs.

TABLE III: (RQ₀): Judge evaluation averaged over 5 runs.

Model	SafeQA		NaviQA-I/II	
	F-1 (Binary)	Time	F-1 (Continuous)	Time
GPT-3.5	0.76	1.0	0.75	0.9
GPT-4O-MINI	0.79	1.1	0.79	1.2
GPT-4O	0.76	1.3	0.77	1.5
GPT-5-CHAT	0.77	1.5	0.78	1.8
DEEPEEK-V2-16B	0.64	1.1	0.66	2.1
DEEPEEK-V3	0.80	1.9	0.40	1.0
MISTRAL 7B	0.76	1.2	0.73	2.3

LLMs under Test. We selected three cloud-based LLMs DEEPEEK-V3, GPT-4O, and GPT-5-CHAT. We had to exclude locally deployed LLMs such as LLAMA 3, QWEN 2.5-7B, DEEPEEK-V2-16B as these LLMs yielded failure rates of over 90% in preliminary experiments conducted.

Testing Setup. We use a search budget of 3 hours, which was determined to be a reasonable duration for preliminary system-level tests based on discussions with test engineers at BMW. Further we use a population size of 20, also selected based on pilot experiments. We use the same and default mutation and crossover parameters as in NaviQA, and use the same setup across all algorithms and systems under comparison. We select constraints based on discussions with BMW experts, and remove, e.g., the price range and food types selection from categories such as *hospital* or *museum*.

F. Results

1) **Judge evaluation (RQ₀):** Table III presents the results for the LLM judge evaluation, for both use cases. In the case of SafeQA, for the binary judgment task, the best-performing LLMs in terms of F-1 values were DEEPEEK-V3 and GPT-4O-MINI. Similarly, for the continuous judgment task, GPT-4O-MINI achieved the best results. Thus, we selected GPT-4O-MINI for integration into STELLAR in the rest of the study also due to its faster inference time.

Regarding NaviQA-I/II, to evaluate the *Fitness Response*, we first generated 30 question-answer pairs using GPT-4O-MINI, guided by the feature dimensions of our framework. Each response was prompted to vary along *R*, *D*, and *P*. Invalid or irrelevant outputs were manually filtered and regenerated. We then conducted a human study with 10 participants from BMW, who rated each response on *R*, *D*, *P*, and provided an overall binary judgment *O*. A total of 300 annotations were collected under the same evaluation conditions used by the LLM judge. An excerpt of the questionnaire is available in the replication package [34]. Following established methodology [55], we assessed (1) inter-rater agreement via Fleiss' Kappa [56], and (2) the alignment between LLM judgments and majority-vote human labels. Substantial agreement was observed for *R* (0.62) and *P* (0.69), with moderate agreement for *D* (0.43). The lower consistency on *D* reflects its subjective nature, as perceived difficulty depends more on individual interpretation than on directly observable qualities.

Overall, the judge's accuracy achieved F-scores between 0.68 and 0.76 across evaluated models (Table III), whereby multi-sample prompting did not offer significant gains but higher costs. We therefore selected GPT-4O-MINI as our judge, balancing accuracy, latency, and cost. For optimization, we compute a continuous fitness score as $f_1 = w_R \cdot R + w_D \cdot D + w_P \cdot P$. The weights ($w_R = 0.55$, $w_D = 0.3$, $w_P = 0.15$) were first derived via logistic regression on human overall judgments *O* [57] and validated with BMW experts.

RQ₀ (judge evaluation). Among the evaluated models, GPT-4O-MINI demonstrates the best balance between performance and time/cost efficiency across both case studies SafeQA and NaviQA, achieving F1-scores of 0.79 and 0.71, respectively.

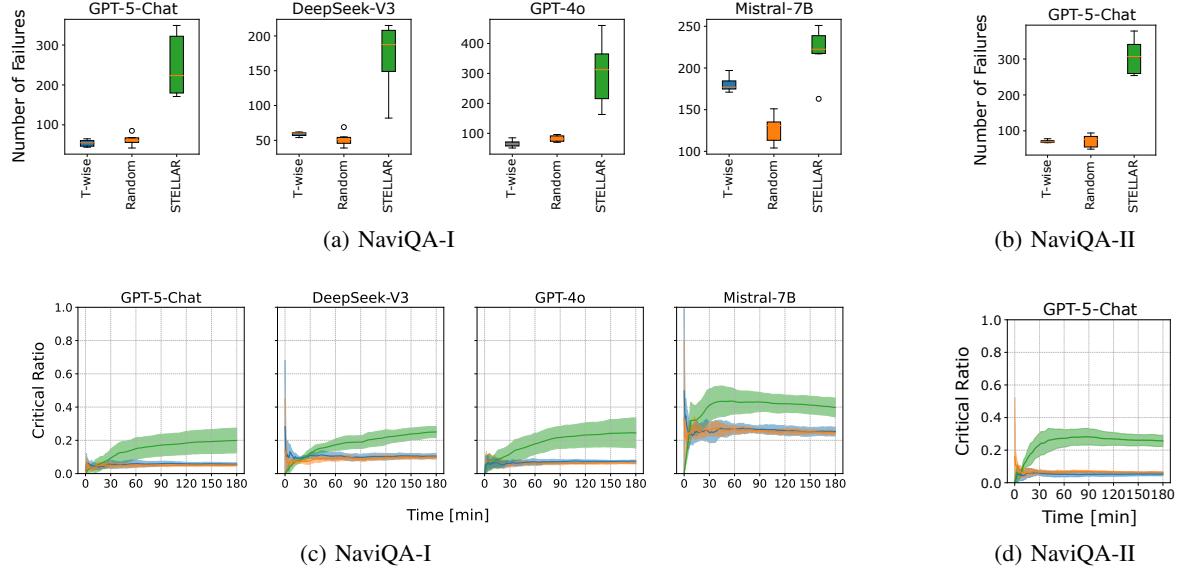


Fig. 3: Results for RQ₁ (NaviQA). (top) Number of failures found by each approach after 3 hours of search time. (bottom) Mean ratio between failures found and generated test cases with standard deviation over 6 runs.

2) *Effectiveness (RQ₁)*: For SafeQA, the failure counts are reported in Figure 2 (top). Across all LLMs, STELLAR consistently identifies more failures than ASTRAL, T-WISE, and RS. The smallest improvement is observed with GPT-4o, while the largest gain occurs with MISTRAL 7B. As expected, the lowest number of failures overall is found for GPT-5-CHAT, which reflects its stronger alignment and safety mechanisms. In general, all techniques detect more failures in smaller, local models compared to large cloud-based models (up to 671B parameters). Between the baseline methods, ASTRAL only outperforms RS and T-WISE in two out of six LLM configurations; otherwise, their performance is largely comparable, indicating limited benefit from coverage-based guidance in SafeQA.

Figure 2 (bottom) reports the failure ratio, i.e., identified failures relative to all executed tests. Across all LLMs, this ratio stabilizes over time, confirming that the allocated search budget is sufficient. STELLAR consistently achieves the highest failure ratio compared to all baselines. The lowest ratio (27%) is observed on GPT-5-CHAT, consistent with its failure counts, while the highest ratio (80%) is recorded on MISTRAL 7B. Overall, STELLAR identifies on average a 2.2 times higher failure rate than the best baseline approach.

For NaviQA, failure counts are shown in Figure 3 (top). As in SafeQA, STELLAR consistently discovers more failures than both RS and T-WISE across all AUTs. The highest failure counts are observed with MISTRAL 7B, which is consistent with its smaller model size (7B). STELLAR also exhibits greater variability, indicating broader exploration. In NaviQA-II, STELLAR clearly outperforms the baselines in both absolute failures and failure ratios.

Failure rates are reported in Figure 3 (bottom). Across all AUTs, failure ratios converge over time, confirming sufficient testing budget. With STELLAR, GPT-5-CHAT achieves the

lowest rate (24%), followed by DEEPSEEK-V3 and GPT-4o, while MISTRAL 7B reaches the highest failure rate (39%). Overall comparison, STELLAR achieves, on average, a 2.5 times higher failure rate. To assess statistical significance, we applied the Mann–Whitney U test [58] ($\alpha = 0.05$) and quantified effect size using Vargha–Delaney A^{12} [59]. The improvements of STELLAR over all baselines are statistically significant, with large effect sizes, for both case studies and the three AUTs.

RQ₁ (effectiveness). STELLAR detects significantly more failures and produces higher failure rates than randomized/combinatorial testing, and a state-of-the-art approach in safety testing of six LLMs. A second study with two AUT types and three LLM variants similarly demonstrates that STELLAR outperforms randomized/combinatorial testing in identifying failures.

3) *Diversity (RQ₂)*: Table IV presents the failure diversity results based on coverage. For SafeQA, STELLAR consistently surpasses ASTRAL across all models. Although RS and T-WISE occasionally achieve competitive coverage, these cases occur primarily on weaker AUTs, where failures are easier to trigger and diversify. As a result, their diversity scores are less indicative of true behavioral robustness. In contrast, on harder-to-test models such as GPT-5-CHAT, where failure modes are significantly harder to expose (based on the results in Figure 2, STELLAR achieves the highest coverage (up to 98%), demonstrating its strength. This highlights that STELLAR maintains diversity not through random dispersion, but through targeted exploration of challenging failure regions. For both NaviQA-I and NaviQA-II, STELLAR matches or exceeds the coverage achieved by RS and T-WISE across all AUTs. In SafeQA, coverage differences between STELLAR

TABLE IV: Results for RQ₂ (diversity). Coverage across failure clusters (%), averaged over 10 runs).

Case Study	Model	STELLAR	ASTRAL	T-WISE	RS
SafeQA	DEEPSEEK-V2-16B	79	56	89	89
	MISTRAL 7B	82	58	90	92
	DEEPSEEK-V3	79	65	88	89
	GPT-5-CHAT	98	93	96	96
	QWEN 3-8B	86	61	91	91
	GPT-4o	93	78	91	93
NaviQA-I	MISTRAL 7B	100	–	100	100
	DEEPSEEK-V3	100	–	100	100
	GPT-5-CHAT	96	–	88	87
NaviQA-II	GPT-5-CHAT	99	–	95	96

and ASTRAL are statistically significant, with large effect sizes favoring STELLAR. In NaviQA, no baseline shows a significant advantage over STELLAR. These findings align with prior observations that randomized methods may explore failures more broadly than optimization-based testing [60], [43]. However, STELLAR maintains competitive diversity while achieving substantially higher failure discovery (RQ₁). Since STELLAR does not explicitly target diversity, exploring such strategies remains an open direction for future work.

RQ₂ (diversity). For the safety case study, STELLAR achieves consistently higher coverage than ASTRAL. Although RS and T-WISE occasionally match coverage, this occurs only on weaker AUTs. On stronger models (e.g., GPT-5-CHAT) STELLAR obtains the highest diversity, confirming its effectiveness. For the navigational case study, STELLAR delivers higher coverage than the baselines.

V. THREATS TO VALIDITY

Internal Validity. Internal validity concerns factors that might affect the correctness of our results. First, several analyses required thresholds (e.g., for clustering and distance metrics). Different threshold values might alter the results. However, we selected these based on preliminary experiments and default values, but other choices could lead to different outcomes. Second, LLMs are inherently non-deterministic, producing different outputs for the same prompt. Additionally, API call durations can vary. We mitigated this by repeating runs and averaging results, but some residual randomness may persist. Finally, the LLM-based generator may occasionally produce utterances that do not exactly reflect the intended feature values, introducing noise in the test set. We evaluate the generator initially and monitor such cases.

Construct validity. Construct validity concerns the degree to which our measures and constructs reflect the phenomena of interest. First, we measure convergence via the ratio of discovered failures. Embedding distance checks help cluster related inputs, yet they cannot account for every possible variation. Second, in our case studies, we defined three custom judge dimensions with the help of domain experts and user data. While expert input improves realism, it also introduces

subjectivity. We partially mitigated this by conducting correlation analysis with human evaluators, though replication with additional experts would further strengthen validity. In addition, we conducted a human evaluation to assess the accuracy of the LLM-generated test inputs. Finally, we relied on ASTRAL in an offline RAG configuration. While this setup ensures repeatability, it may not capture the performance of an OpenAI-based online RAG system.

External Validity. External validity concerns the extent to which our results can be generalized. We studied two different case studies involving three AUTs and up to six LLMs. Although the systems and models differ in nature, the scope is still limited. Generalization to other domains, larger systems, or additional LLM families should be made with caution.

VI. QUALITATIVE EVALUATION

We performed a qualitative evaluation for the study NaviQA-II. In particular, we clustered using k-means all failures found by all approaches based on embeddings of system answers received. It should be noted, that this clustering differs from the analyses in RQ₂, as it is based on the system outputs rather than on test inputs. Then we sampled 100 failures per cluster and manually assigned each cluster to a specific failure type. In addition, we measured per cluster how many failures of each cluster are identified by STELLAR or by remaining approaches and reported all results in Table V.

The identified failures range from endpoint and retrieval errors to misinterpretations caused by linguistic perturbations. We then interviewed a domain expert with 7 years of working experience in the testing of AI-enabled systems such as NaviQA-II and presented him the failures found and the types extracted. We then asked the following questions:

Q1: Are the failures consistent with the extracted failure types, and are the failure types realistic? Following the expert, all the identified failure types are in line with the inspected failures, and can occur in NaviQA-II.

Q2: Which of the failure types have already been witnessed when testing NaviQA-II before? Based on the response, failure types F1, F2, F6, F7, F8, and F9 have already been frequently detected, F4 has been seen in different contexts but not yet in this expression (e.g., so far it only occurred when German street names were pronounced based on English pronunciation logic by an English native). Only the failure types F3 and F5 have not been detected so far through testing.

Q3: What is the severity of the identified failures/types? The majority of the identified failures are of high severity, highlighting the importance of preventing their propagation to the end user. Failure F6 is of low severity, as this type is difficult to avoid due to likely synchronization issues within the system. Based on the interview, we can confirm that our testing uncovered a wide spectrum of distinct failure types. Especially, STELLAR achieved high detection rates in categories such as F3 and F5, which the domain expert identified as particularly challenging to reveal. This suggests that STELLAR is especially effective at uncovering uncommon failures that are likely to be missed by baseline methods.

TABLE V: Observed failure types in NaviQA-II with criticality and ratio of failures identified by STELLAR per type.

ID	Failure Type	Failure Description	Example Answer	Criticality	Ratio (%)
F1	Endpoint Failure	AUT endpoint call fails.	—	High	91
F2	Incorrect Rating	POI is returned but has an incorrect rating.	—	Medium	79
F3	Name Misinterpretation	Perturbation of the utterance causes the system to misinterpret a POI name constraint.	“I could not find ae supermarket.”	Medium	83
F4	Language Misclassification	Perturbation of the utterance causes the system to incorrectly detect a language change.	“Schau me please a bar” → “Habe nichts passendes gefunden.”	High	86
F5	Technical Output	System outputs technical information not intended for the user.	“You should apply the filter ‘payment:card’...”	High	60
F6	Search Not Performed	System does not perform the search but repeats the request constraints.	“Let me search a German bar.”	Low	48
F7	POI Retrieval	Requested POI exists but cannot be retrieved.	“Hmm, no Thai cafes nearby. Want me to look for Thai restaurants?”	High	84
F8	Wrong Intent	The system misunderstands the intent of the request.	“I’m sorry, I cannot answer to that.”	High	64
F9	Empty Output	System returns empty text.	—	Medium	57

VII. RELATED WORK

Several datasets have been developed to evaluate LLMs across different application domains. For safety testing, representative examples include BeaverTails [15], Do-Not-Answer [61], and ToxiGen [62]. For holistic evaluation of general capabilities, benchmarks such as MMLU [63] and BIG-Bench [55] are commonly used. Furthermore, to assess conversational systems, multiple dialogue-oriented datasets have been introduced, including CoQA [64], MMDialog [65], VACW [66], MultiWOZ 2.2 [14], and KVRET [67], which particularly focus on tasks involving navigational requests and recommendations. However, these datasets may already be included in the training data of the LLM under test, which undermines their effectiveness for evaluation.

To address this challenge, several automated testing approaches have been proposed that generate variations or build upon existing datasets. In the following, we describe the main approaches and compare them with STELLAR. For instance, Andriushchenko et al. [49] tried to jailbreak LLMs by modifying a suffix that is appended to a prompt template, while the suffix content is selected based on a randomized search. They achieved up to 100% attack rates on leading safety-aligned LLMs. However, this approach is tailored to robustness testing of LLMs, while STELLAR is generic, which also considers functional testing of LLM applications. METAL [52] is a metamorphic testing framework that benchmarks LLMs across quality attributes such as robustness and fairness. However, it relies on the availability of an initial, diverse dataset where both inputs and expected outputs are already defined. Moreover, its testing process is static, as it does not incorporate a feedback loop to guide the generation of likely failing test cases. In contrast, STELLAR actively modifies test inputs to dynamically explore failure-inducing behaviors.

MORTAR [68] is a framework to benchmark LLM-based dialogue systems by applying operations on the turn level, such as deletion, repetition, or shuffling of turns. Failures can be detected by evaluating outcomes and applying metamorphic relations. Also, this approach requires the existence of an initial dataset to compare the generated responses with. Yoon

et al. [69] presented an adaptive randomized test selection approach that requires an initial dataset, but selects likely failure-revealing test inputs based on evaluations performed on existing data. This is done by computing the distance of a candidate test input to already generated test inputs, prioritizing test inputs with a higher distance. EvoTox [70] focuses on testing LLMs’ toxicity, providing a systematic way to quantitatively surface toxic responses. ASTRAL [17] is the most closely related automated test generation approach, and we include it in our empirical comparison. Similar to STELLAR, it discretizes the search domain and constructs a coverage matrix over feature combinations. However, ASTRAL faces two key limitations: it does not scale well to a high number of features, and it lacks a feedback loop from evaluated test inputs. In contrast, STELLAR incorporates an optimization process that leverages feedback to guide the generation of new, potentially failing test cases.

VIII. CONCLUSION

In this paper, we introduced STELLAR, a search-based testing framework for benchmarking LLM-based applications. Through two case studies and three systems under test, covering both malicious input handling and task-oriented dialogues, we demonstrated that STELLAR consistently uncovers significantly more failures than randomization/combinatorial-based methods and a state-of-the-art automated testing baseline. In our future work, we plan to extend STELLAR to applications involving multiple modalities, such as image and audio inputs, and to investigate its applicability to multi-turn dialogue scenarios, where interaction dynamics and conversational state introduce new challenges for testing.

IX. DATA AVAILABILITY

The pipeline used to obtain the results discussed in this work and the results are available in our replication package [34].

ACKNOWLEDGEMENTS

This research was funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy, and by the BMW Group. We thank all survey participants and colleagues from BMW who supported this work.

REFERENCES

- [1] Y. Zhang, H. Jin, D. Meng, J. Wang, and J. Tan, “A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods,” 2025.
- [2] D. Elshin, N. Karpachev, B. Gruzdev, I. Golovanov, G. Ivanov, A. Antonov, N. Skachkov, E. Latypova, V. Layner, E. Enikeeva, D. Popov, A. Chekashev, V. Negodin, V. Frantsuzova, A. Chernyshev, and K. Denisov, “From general LLM to translation: How we dramatically improve translation quality using human evaluation data for LLM finetuning,” in *Proceedings of the Ninth Conference on Machine Translation*, 2024.
- [3] Duolingo. (2024) Duolingo max with gpt-4.
- [4] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, “A survey on code generation with llm-based agents,” 2025.
- [5] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, “From llms to llm-based agents for software engineering: A survey of current, challenges and future,” 2025.
- [6] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, “Large language model-based agents for software engineering: A survey,” 2024.
- [7] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, “Llms in software security: A survey of vulnerability detection techniques and insights,” *ACM Comput. Surv.*, 2025.
- [8] K. E. Friedl, A. G. Khan, S. R. Sahoo, M. R. A. H. Rony, J. Germies, and C. Süß, “Inca: Rethinking in-car conversational system assessment leveraging large language models,” 2023.
- [9] R. Giebisch, K. E. Friedl, L. Sorokin, and A. Stocco, “Automated factual benchmarking for in-car conversational systems using large language models,” in *Proceedings of the 36th IEEE Intelligent Vehicles Symposium*, ser. IV ’25, 2025.
- [10] B. S. Ahmed, L. O. Baader, F. Bayram, S. Jagstedt, and P. Magnusson, “Quality Assurance for LLM-RAG Systems: Empirical Insights from Tourism Application Testing,” in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2025.
- [11] R. G. Rapisarda, D. Ginelli, D. Clerissi, and L. Mariani, “Test Case Generation for Dialogflow Task-Based Chatbots,” in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2025.
- [12] BMW Group Press Release. (2024) Generative AI, Augmented Reality and Teleoperated Parking – The Digital Experience in the BMW of the Future at the Consumer Electronics Show (CES) 2024. BMW Group. Accessed: 2025-10-17. [Online]. Available: <https://tinyurl.com/3cbapys8>
- [13] T. Xie, X. Qi, Y. Zeng, Y. Huang, U. M. Schwag, K. Huang, L. He, B. Wei, D. Li, Y. Sheng, R. Jia, B. Li, K. Li, D. Chen, P. Henderson, and P. Mittal, “SORRY-bench: Systematically evaluating large language model safety refusal,” in *The Thirteenth International Conference on Learning Representations*, 2025.
- [14] X. Zang, A. Rastogi, S. Sunkara, R. Gupta, J. Zhang, and J. Chen, “Multiwoz 2.2: A dialogue dataset with additional annotation corrections and state tracking baselines,” in *Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI, ACL 2020*, 2020.
- [15] J. Ji, M. Liu, J. Dai, X. Pan, C. Zhang, C. Bian, C. Zhang, R. Sun, Y. Wang, and Y. Yang, “Beavertails: Towards improved safety alignment of llm via a human-preference dataset,” 2023.
- [16] J. S. Bradbury and R. More, “Addressing Data Leakage in HumanEval Using Combinatorial Test Design,” in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025.
- [17] M. Ugarte, P. Valle, J. A. Parejo, S. Segura, and A. Arrieta, “Astral: A tool for the automated safety testing of large language models,” in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA Companion ’25, 2025.
- [18] A. Zeller, “Search-based testing and system testing: A marriage in heaven,” ser. SBST, 2017.
- [19] P. McMinn, “Search-based software testing: Past, present and future,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011.
- [20] V. Riccio and P. Tonella, “Model-based exploration of the frontier of behaviours for deep learning system testing,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.
- [21] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, “Testing Machine Learning based Systems: A Systematic Mapping,” *Empirical Software Engineering*, 2020.
- [22] O. Weißl, A. Abdellatif, X. Chen, G. Merabishvili, V. Riccio, S. Kacianka, and A. Stocco, “Targeted deep learning system boundary testing,” *ACM Transactions on Software Engineering and Methodology*, 2025.
- [23] A. Ramírez, J. R. Romero, and S. Ventura, “A survey of many-objective optimisation in search-based software engineering,” *Journal of Systems and Software*, vol. 149, 2019.
- [24] S. Nejati, L. Sorokin, D. Safin, F. Formica, M. M. Mahboob, and C. Menghi, “Reflections on surrogate-assisted search-based testing: A taxonomy and two replication studies based on industrial ADAS and simulink models,” *Inf. Softw. Technol.*, vol. 163, 2023.
- [25] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing vision-based control systems using learnable evolutionary algorithms,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.
- [26] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, “Testing autonomous cars for feature interaction failures using many-objective search,” in *Proceedings of ASE ’18*, ser. ASE 2018, 2018.
- [27] L. Sorokin, T. Munaro, D. Safin, B. H.-C. Liao, and A. Molin, “OpenSBT: A modular framework for search-based testing of automated driving systems,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24, 2024.
- [28] L. Sorokin and N. Kerscher, “Guiding the search towards failure-inducing test inputs using support vector machines,” in *Proceedings of the 5th IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning*, ser. DeepTest ’24, 2024.
- [29] X. Chen, M. Biagiola, V. Riccio, M. d’Amorim, and A. Stocco, “XMutant: XAI-based Fuzzing for Deep Learning Systems,” *Empirical Software Engineering*, 2025.
- [30] Maryam, M. Biagiola, A. Stocco, and V. Riccio, “Benchmarking generative ai models for deep learning test input generation,” in *Proceedings of the 18th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST ’25. IEEE, 2025, p. 12 pages.
- [31] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, “Misbehaviour prediction for autonomous driving systems,” in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE ’20. ACM, jun 2020.
- [32] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, “Connecting large language models with evolutionary algorithms yields powerful prompt optimizers,” in *The Twelfth International Conference on Learning Representations*, 2024.
- [33] R. Pryzant, D. Iter, J. Li, Y. Lee, C. Zhu, and M. Zeng, “Automatic prompt optimization with “gradient descent” and beam search,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.
- [34] Anonymous, “Replication package,” <https://github.com/stellar-paper/stellar.git>.
- [35] M. R. A. H. Rony, C. Suess, S. R. Bhat, V. Sudhi, J. Schneider, M. Vogel, R. Teucher, K. Friedl, and S. Sahoo, “CarExpert: Leveraging large language models for in-car conversational question answering,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2023.
- [36] S. Alharbi, M. Alrazgan, A. Alrashed, T. Alnomasi, R. Almojil, R. Alharbi, S. Alharbi, S. Alturki, F. Alshehri, and M. Almojil, “Automatic speech recognition: Systematic literature review,” *IEEE Access*, vol. 9, 2021.
- [37] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out*, 2004.
- [38] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002.
- [39] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, 2002.
- [40] K. Deb, K. Sindhya, and T. Okabe, “Self-adaptive simulated binary crossover for real-parameter optimization,” in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO ’07, 2007.

- [41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” 2013.
- [42] “Llamaindex,” https://developers.llamaindex.ai/python-framework-api-reference/evaluation/semantic_similarity/.
- [43] L. Sorokin, D. Safin, and S. Nejati, “Can search-based testing with pareto optimization effectively cover failure-revealing test inputs?” *Empirical Software Engineering*, vol. 30, no. 1, 2024.
- [44] T. Munaro, M. Turalija, S. Barner, and M. Halak, “A systematic approach to fault injection test case generation in practice,” in *Software Engineering and Advanced Applications*, 2026.
- [45] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing vision-based control systems using learnable evolutionary algorithms,” *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.
- [46] M. Biagiola and P. Tonella, “Testing of deep reinforcement learning agents with surrogate models,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, 2024.
- [47] R. Feldt, S. Poulding, D. Clark, and S. Yoo, “Test set diameter: Quantifying the diversity of sets of test cases,” in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.
- [48] J. Wen, P. Ke, H. Sun, Z. Zhang, C. Li, J. Bai, and M. Huang, “Unveiling the implicit toxicity in large language models,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.
- [49] M. Andriushchenko, F. Croce, and N. Flammarion, “Jailbreaking leading safety-aligned LLMs with simple adaptive attacks,” 2025.
- [50] N. Asghar, “Yelp dataset challenge: Review rating prediction,” *CoRR*, vol. abs/1605.05362, 2016.
- [51] K. R. Shahapure and C. Nicholas, “Cluster quality analysis using silhouette score,” in *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, 2020.
- [52] S. Hyun, M. Guo, and M. A. Babar, “METAL: Metamorphic Testing Framework for Analyzing Large-Language Model Qualities ,” in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024.
- [53] “Cmu-pronouncing library,” 2022.
- [54] H. Bortfeld, S. Leon, J. Bloom, M. Schober, and S. Brennan, “Disfluency rates in conversation: Effects of age, relationship, topic, role, and gender.” *Language and speech*, vol. 44, 2001.
- [55] G. Bai, J. Liu, X. Bu, Y. He, J. Liu, Z. Zhou, Z. Lin, W. Su, T. Ge, B. Zheng *et al.*, “Mt-bench-101: A fine-grained benchmark for evaluating large language models in multi-turn dialogues,” *arXiv preprint arXiv:2402.14762*, 2024.
- [56] J. L. Fleiss, “Measuring nominal scale agreement among many raters.” *Psychological bulletin*, vol. 76, no. 5, 1971.
- [57] A. Cabrera, “Logistic regression analysis in higher education: An applied perspective,” *Higher education: Handbook of theory and research X*/Agathon Press, 1994.
- [58] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, 1945.
- [59] A. Vargha and H. D. Delaney, “A critique and improvement of the common language effect size statistics of mcgraw and wong.” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, 2000.
- [60] D. Humeniuk, F. Khomh, and G. Antoniol, “Ambiegen: A search-based framework for autonomous systems testingimage 1,” *Science of Computer Programming*, vol. 230, 2023.
- [61] Y. Wang, H. Li, X. Han, P. Nakov, and T. Baldwin, “Do-not-answer: Evaluating safeguards in LLMs.” in *Findings of the Association for Computational Linguistics: EACL 2024*, 2024.
- [62] T. Hartvigsen, S. Gabriel, H. Palangi, M. Sap, D. Ray, and E. Kamar, “ToxiGen: A large-scale machine-generated dataset for adversarial and implicit hate speech detection,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022.
- [63] Y. Wang, X. Ma, G. Zhang, Y. Ni, A. Chandra, S. Guo, W. Ren, A. Arulraj, X. He, Z. Jiang *et al.*, “Mmlu-pro: A more robust and challenging multi-task language understanding benchmark,” *arXiv preprint arXiv:2406.01574*, 2024.
- [64] S. Reddy, D. Chen, and C. D. Manning, “Coqa: A conversational question answering challenge,” *Transactions of the Association for Computational Linguistics*, vol. 7, 2019.
- [65] J. Feng, Q. Sun, C. Xu, P. Zhao, Y. Yang, C. Tao, D. Zhao, and Q. Lin, “Mmdialog: A large-scale multi-turn dialogue dataset towards multi-modal open-domain conversation,” *arXiv preprint arXiv:2211.05719*, 2022.
- [66] I. Siegert, ““Alexa in the wild” – collecting unconstrained conversations with a modern voice assistant in a public environment,” in *Proceedings of the Twelfth Language Resources and Evaluation Conference*, 2020.
- [67] M. Eric, L. Krishnan, F. Charette, and C. D. Manning, “Key-value retrieval networks for task-oriented dialogue,” in *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue (SIGDIAL)*, 2017.
- [68] G. Guo, A. Aleti, N. Neelofar, and C. Tantithamthavorn, “Mortar: Metamorphic multi-turn testing for llm-based dialogue systems,” 2024.
- [69] J. Yoon, R. Feldt, and S. Yoo, “ Adaptive Testing for LLM-Based Applications: A Diversity-Based Approach ,” in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2025.
- [70] S. Corbo, L. Bancale, V. D. Gennaro, L. Lestingi, V. Scotti, and M. Camilli, “How toxic can you get? search-based toxicity testing for large language models,” *IEEE Transactions on Software Engineering*, vol. 51, no. 11, pp. 3056–3071, 2025.